CSC 543 Multiprocessing & Concurrent Programming, Spring 2023

Dr. Dale E. Parson, Assignment 3, latches, barriers, atomic arrays, thread pools, etc.
This assignment is due by 11:59 PM on Saturday April 8 via make turnitin.

This assignment is a re-do of parts of both assignments 1 and 2 to use new mechanisms out of the Java concurrency library that we have not used before. The source file comments labeled with **STUDENT** give the details.

Perform the following steps to set up this project and to get my handout. Start out in your login directory on csit (a.k.a. acad).

```
cd  $HOME
cp  ~parson/multip/PipesPrisoners2023.problem.zip multip/PipesPrisoners2023.problem.zip
```

Use machine **mcgonagall** for development and testing. Run **make turnitin** on acad or mcgonagall by the due date.

After logging into mcgonagall, do the following.

```
cd   ./multip
unzip  PipesPrisoners2023.problem.zip
cd  ./PipesPrisoners2023
make  clean test
```

Running **make clean test** on the handout code passes because it consists of my solutions to Assignments 1 and 2, which constitute your starting point. There are two portions to the assignment, those tested by running **make clean testprison** for testing your updated Assignment 2 (this is the easier part), and running **make clean testpipe** for testing the updated Assignment 1 pipeline (harder). You can do them in either order. Running **make clean test** tests both; **make testtime** collects execution & CPU times on the pipeline. The following source file comments explain the work you have to do. I have some more documentation added with these comments below.

Interactive stack trace on a blocked Java process:

If one of your program hangs while running, you can control-C to kill it and then start it in command line mode using the following Unix command lines. Wait a while until it hangs, and then type control-\ (control-backslash) to see a stack trace of every thread in the process. Look at the main thread and the threads named "Thread-0" etc., ignore the GC (garbage collection), Compile, and other run-time threads run by the JVM.

```
CLASSPATH=..:./jcip-annotations.jar /usr/bin/java PipesPrisoners2023.PrisonerTest implicit

CLASSPATH=..:./jcip-annotations.jar /usr/bin/java PipesPrisoners2023.PrisonerTest explicit

CLASSPATH=..:jcip-annotations.jar .usr/bin/java PipesPrisoners2023.BigDecimalPipelineBuilder true
true PipesPrisoners2023.PipeSourceRandom '12345 3000 3000'
java.util.concurrent.LinkedBlockingDeque PipesPrisoners2023.PipeStageMath +
java.util.concurrent.LinkedBlockingDeque PipesPrisoners2023.PipeSinkFile sink.out
```

Use the pipeline command line for the hanging test run that appears on your screen.

********** **PRISONERS' DILEMMA:**

Source code changes to the updated **make clean testprison**:

PrisonerTest.java

```
33      // STUDENT CSC543 ASSN3 #1: For the IMPLICIT players, construct
34      // a CountDownLatch object, initialized to the appropriate count for
35      // this main thread + ALL of the player threads, and have each
36      // of them enter that CountDownLatch object in the appropriate
37      // place in the code, instead of of the join() loop in PrisonerTest.
38      // For the EXPLICIT players, use a CyclicBarrier in the same way.

49      // STUDENT CSC543 ASSN3 #1: Get rid of this join() for-loop.
50      // See #1 instructions above.
```

PlayerImplicit.java

```
// STUDENT CSC543 ASSN3 #1: See PrisonerTest.java IMPLICIT section
// of CSC543 ASSN3 #1 STUDENT comments. Remove the conditional
// .join() call below, since both player 0 and player 1 must enter
// the CountDownLatch instead of joining.
// STUDENT CSC543 ASSN3 #2: Replace all uses of the intrinsic lock
// and its condition variable on messageBuffer access below,
// and replace the String [] messageBuffer itself, with a 2-element
// java.util.concurrent.atomic.AtomicReferenceArray<String> object
// field called messageBuffer that is used in an equivalent way.
// There will be no actual locking-waiting. Instead, a player must
// spin in a loop until the messageBuffer into which it is writing is null,
// and it must spin in a loop until the messageBuffer from which it is
// reading is non-null. The basic logic is the same, but it uses
// spin loops instead of locks-and-waits. Make sure it remains thread-safe.
// Make sure to update the @GuardedBy annotation if necessary.
```

PlayerExplicit.java

```
// STUDENT CSC543 ASSN3 #1: See PrisonerTest.java EXPLICIT section
// of CSC543 ASSN3 #1 STUDENT comments. Remove the conditional
// .join() call below, since both player 0 and player 1 must enter
// the CyclicBarrier instead of joining.
// STUDENT CSC543 ASSN3 #3: Replace all uses of the extrinsic lock
// and its condition variable on messageBuffer access below,
// with two two-element arrays of java.util.concurrent.Semaphore
// objects called isNull (initialized to a count of 1 and fair = true),
// and isReady (initialized to a count of 0 and fair = true).
// State "sendMyAction" must acquire the correct isNull element,
// then write a message, then release the correct isReady element.
// State "awaitOtherAction" must acquire the correct isReady element,
// then read and null a message slot, then release the correct isNull
```

// element. In this case the String [] messageBuffer is @GuardedBy
// both Semaphores; one guards writing, and the other guards
// reading/nulling. Semaphore-based approaches predate condition
// variables, and don't quite fit the @GuardedBy semantics.

WARNING, this join near the bottom of PlayerImplicit and PlayerExplicit never run because of the while loop condition:

```
case terminated :
    if (partner != null) {
       partner.join();
    }
    return ;
```

Please put your termination code that uses CountDownLatch or CyclicBarrier down in the finally block at the bottom.

********** **PIPELINE**:

Source code changes to the updated **make clean testpipe**:

StudentBlockingQueue.java (see additional comments below)

See STUDENT comments in **StudentBlockingQueue.java**. DO NOT CHANGE LinkedList<E> IN container IN THIS LINE OF CODE:

LinkedList<E> container = new LinkedList<E>();

You must continue to use a LinkedList<E> object while making StudentBlockingQueue thread-safe.

All of the methods following the empty constructor in that class were generated by running **make genstudent** which runs Jython script **GenQueueMethods.py** to create file **StudentBlockingQueue.start.txt**. Per instructions in **StudentBlockingQueue.java** comments, you must run:

./genstudent YourLockPrefix YourLockSuffix

Place single quotes around each of YourLockPrefix and YourLockSuffix
to use literal characters such as ( ) which the shell would otherwise interpret.
You can use an empty string " as one of the arguments to genstudent if you
Want to substitute a black.

This reads StudentBlockingQueue.java.template and writes junk.java with a replacement for each PREFIX and SUFFIX.

For a function definition in StudentBlockingQueue.java.template such as this:

```
public boolean add(
    E arg0
) {
    {   //STUDENT_PREFIX template
```

```
        return container.add(arg0);
    }   //STUDENT_SUFFIX template
}
```

Running **./genstudent '(BEFORE)' '(AFTER)'**

Creates this function in junk.java.

```
public boolean add(
    E arg0
) {
    (BEFORE) {
        return container.add(arg0);
    } (AFTER)
}
```

All of the non-blocking functions from interface Queue delegate by calling the underlying LinkedList<E> container method. You must update STUDENT_PREFIX and STUDENT_SUFFIX (you can use an empty string " for replacement) via ./**genstudent**. Then inspect junk.java, and if you feel it is correct, move it to StudentBlockingQueue.java before hand editing that file.

mv junk.java StudentBlockingQueue.java

The generated BlockingQueue blocking methods must be completed via hand coding, using a condition variable to implement wait()ing where necessary.

```
public E take(
) throws java.lang.InterruptedException {
    /* STUDENT BlockingQueue problem */
    return null ;
}
```

From the handout StudentBlockingQueue.java:

```
*   When implementing the BlockingQueue blocking functions, STUDENTs
 *  may assume that the offer() and put() calls will always succeed,
 *  since LinkedList does not have a fixed bound on size. Also,
 *  where TimeUnit appears, just ignore it and return a result
 *  without waiting.
 *
 *  Provide the correct field modifiers for "container" same as in
 *  previous assignments, and provide all required Annotations.
 *
 *  Ignore this warning from the compiler:
 *  Note: StudentBlockingQueue.java uses unchecked or unsafe operations.
 *  Note: Recompile with -Xlint:unchecked for details.
 *
 *  After the above is done, uncomment the following three lines in testscript:
 *  runatest false false PipesPrisoners2023.StudentBlockingQueue
 *  runatest true false PipesPrisoners2023.StudentBlockingQueue
 *  runatest true true PipesPrisoners2023.StudentBlockingQueue
```

Note that I have annotated generated blocking functions with the comment "STUDENT BlockingQueue problem". For any BlockingQueue operations that must block when **READING** an empty queue, use the condition variable associated with your locking mechanism to wait until contents appear. For any BlockingQueue operations that must block when **WRITING** a queue, assume that LinkedList can complete that write operation without failing. Also, do not catch InterruptedException within StudentBlockingQueue methods; just let it propagate out to the callers to your class. EVERY method that adds data to the container, whether blocking or not, must do the notifyAll(), signalAll(), or your equivalent to wake up any blocked reader threads.

You can ignore the TimeUnit associated with time-limited blocking calls. We are not using any of them in our test framework, and I don't want to complicate things any further.

NOTE:

After you uncomment the runatest lines on StudentBlockQueue in testscript, **make testtime** will fail with the handout testtime.ref because StudentBlockingQueue is not in that reference file.

```
$ make testtime
…
echo "STATISTICS in testtime.txt:"
STATISTICS in testtime.txt:
cut -d'|' -f2-3,6 testtime.txt | sort > testtime.out2
diff testtime.out2 testtime.ref > testtime.dif
make: *** [testtime] Error 1
```

Look at testtime.dif to see the difference. If it is simply absence of StudentBlockingQueue, do the following:

```
$ cat testtime.dif
7d6
< false|false|PipesPrisoners2023.StudentBlockingQueue
13d11
< true|false|PipesPrisoners2023.StudentBlockingQueue
18d15
< true|true|PipesPrisoners2023.StudentBlockingQueue
```

```
$ mv testtime.out2 testtime.ref
```

At this point **make testtime** should work.

****

testscript

```
#
# After StudentBlockingQueue is thread-safe and complete,
# uncomment the following three lines in testscript:
# runatest false false PipesPrisoners2023.StudentBlockingQueue
# runatest true false PipesPrisoners2023.StudentBlockingQueue
# runatest true true PipesPrisoners2023.StudentBlockingQueue
```

Running **make testpipe** tests the pipeline and **make testtime** benchmarks pipeline running times as in Assignment 1.

Visually check your classes to make sure their field declarations and scopes of their locks are thread safe and the classes are properly annotated, included GuardedBy. Testing is not enough by itself. Update thread safety annotations where needed.

After a final visual check and a successful run of **make clean test**, run **make turnitin** by the deadline.